Proyecto 02 – DSA para Downscaling por Interpolación Bilineal Paralela

Avance 3: Implementación base (secuencial)

Gabriel Guzmán Rojas

Sebastián Hernández Bonilla

José Vargas Torres

14 de noviembre de 2025

Índice

1. Implementación base en modo secuencial

1.1. Módulo `ModoSecuencial`

Para el avance 3 se implementó la interpolación. Este módulo recibe, en cada ciclo de reloj, los cuatro píxeles vecinos `I00`, `I10`, `I01` e `I11` (cada uno de 8 bits sin signo) junto con los pesos fraccionarios `alpha` y `beta` en formato Q0.8. A partir de esos valores calcula un píxel interpolado utilizando el algoritmo de interpolación bilineal y entrega el resultado en la salida `pixel_out`. La interfaz también incluye las señales de control `clk`, `rst`, `valid_in` y `valid_out`.

La idea es modelar la siguiente ecuación de interpolación bilineal, separada en dos interpolaciones lineales, primero en el eje x y luego en el eje y:

$$\begin{aligned} a &= I_{00} + \alpha \cdot (I_{10} - I_{00}) \\ b &= I_{01} + \alpha \cdot (I_{11} - I_{01}) \\ v &= a + \beta \cdot (b - a) \end{aligned} \tag{1}$$

donde α y β representan las fracciones horizontales y verticales dentro del cuadrante formado por los cuatro píxeles vecinos.

Formato numérico

Los píxeles de entrada llegan como enteros sin signo de 8 bits (rango 0–255), mientras que `alpha` y `beta` llegan como Q0.8, es decir, un entero de 8 bits donde el valor real se interpreta como `valor/256`. Dentro del módulo todos los cálculos se realizan en Q8.8 (16 bits: 8 bits de parte entera y 8 de parte fraccionaria), de manera que se pueda manejar la parte fraccionaria sin recurrir a punto flotante.

1.2. Flujo interno del datapath

El diseño de `ModoSecuencial` se organizó en cinco etapas lógicas:

1. **Etapa 1: Extensión a Q8.8.** En un bloque `always_comb` se extienden los píxeles y los pesos al formato Q8.8:
   - Para cada píxel se hace un "shift" implícito de 8 bits concatenando ceros en la parte fraccionaria: `I00_q = {I00, 8'h00};`. Esto equivale a $I00 \cdot 2^8$.
   - Para `alpha` y `beta` se concatenan 8 bits de parte entera en cero: `alpha_q = {8'h00, alpha};`, de forma que un valor Q0.8 se ve como Q8.8 con parte entera cero.
2. **Etapa 2: Interpolación horizontal.** En esta etapa se calculan las dos interpolaciones horizontales a y b. Primero se obtienen las diferencias `diff_x0 = I10_q - I00_q` y `diff_x1 = I11_q - I01_q` usando registros con signo de 17 bits, porque la diferencia puede ser negativa. Luego se multiplica cada diferencia por `alpha_q`. El producto se

hace en 32 bits en formato Q16.16. Para regresar a Q8.8 se toman los bits `[23:8]` del resultado, lo que equivale a hacer un corrimiento a la derecha de 8 bits. Dependiendo del signo de la diferencia se suma o se resta el término corregido al píxel original, de forma equivalente a:

$$a_q = I00\_q + \alpha\_q \cdot (I10\_q - I00\_q), \qquad b_q = I01\_q + \alpha\_q \cdot (I11\_q - I01\_q).$$

Esta lógica se implementa con `if (diff_x0 <0)` para manejar correctamente el signo sin perder precisión.

3. **Etapa 3: Interpolación vertical.** Una vez calculados `a_q` y `b_q` se repite el mismo patrón de cálculo, pero ahora en el eje vertical. Se calcula la diferencia `diff_y = b_q - a_q`, se multiplica por `beta_q` y se suma o resta al valor `a_q`:

$$v_q = a_q + \beta_q \cdot (b_q - a_q).$$

Igual que antes, el producto se maneja en Q16.16 y luego se recorta a Q8.8 usando los bits `[23:8]` del resultado de la multiplicación.

4. **Etapa 4: Redondeo y saturación a 8 bits.** El valor interpolado `v_q` está en Q8.8. Para convertirlo a un entero de 8 bits se realiza:

   - **Redondeo**: se suma el valor `0x80` (que corresponde a 0.5 en Q8.8) antes de truncar. Esto se implementa como `v_rounded = {1'b0, v_q} + 17'h0080;`.
   - **Extracción de parte entera**: se toman los bits `[16:8]` en un registro de 9 bits, `pixel_int`, para tener un rango intermedio de 0–511.
   - **Saturación**: si `pixel_int` es mayor a 255 se fuerza a 255; en caso contrario se copian los 8 bits menos significativos a `pixel_clamped`. De esta forma se garantiza que la salida siempre queda en el rango 0–255.

5. **Etapa 5: Registro de salida.** Finalmente, en un bloque `always_ff` sensitivo al flanco positivo del reloj y al reset se registran `pixel_out` y `valid_out`.

   - Si `rst` está activo, la salida se limpia a cero y `valid_out` se pone en 0.
   - En caso contrario, `valid_out` sigue directamente a `valid_in` y, cuando `valid_in` es 1, se almacena el valor interpolado `pixel_clamped` en `pixel_out`.

   De esta forma, desde el punto de vista externo, el módulo entrega un píxel por ciclo cuando `valid_in` está activo, cumpliendo con el requisito de modo secuencial (1 píxel/ciclo).

En resumen, `ModoSecuencial` implementa la interpolación bilineal completa para un píxel de salida, usando aritmética de punto fijo Q8.8 y cuidando el manejo de signo, el recorte de bits y la saturación a 8 bits.

2. Testbench unitario `tb_downscale_ModoSecuencial`

2.1. Objetivo del testbench

El módulo `tb_downscale_ModoSecuencial` sirve como banco de pruebas unitario para validar el comportamiento del módulo `ModoSecuencial`. La idea es reproducir, dentro del testbench, una versión de referencia del algoritmo de downscaling que se usó en el avance 1 (en Python) y comparar, píxel por píxel, la salida del hardware contra esta referencia.

El testbench trabaja con una imagen fuente de tamaño 4×4 y genera una imagen destino de 3×3, aplicando el mismo cálculo de coordenadas y pesos que se usó en el código de software. De esta manera se puede verificar que el módulo secuencial produce resultados correctos dentro de un error máximo de ± 1 LSB.

2.2. Configuración de la imagen y del DUT

Al inicio del testbench se definen parámetros para las dimensiones de la imagen:

- `SRC_H = 4` y `SRC_W = 4` para la imagen fuente.
- `DST_H = 3` y `DST_W = 3` para la imagen destino.

La imagen de entrada se almacena en un arreglo bidimensional `image[y][x]` de 8 bits. Para este avance se cargó una imagen sintética con valores crecientes (10, 30, 50, 70, ...), porque facilita visualizar el efecto de la interpolación.

Además, se declaran las señales que se conectan al *Device Under Test* (DUT): reloj, reset, `valid_in`, los cuatro píxeles vecinos (`I00`, `I10`, `I01`, `I11`), los pesos `alpha` y `beta`, y las salidas `valid_out` y `pixel_out`. Luego se instancia el módulo `ModoSecuencial` conectando estas señales.

El reloj se genera con un bloque `always` que invierte `clk` cada 5 ns, lo que equivale a un periodo de 10 ns (100 MHz). El reset se mantiene activo durante algunos ciclos al inicio de la simulación para garantizar que el DUT arranque en un estado conocido.

2.3. Modelo de referencia en el testbench

Dentro del testbench se implementó una función `bilinear_ref_pixel(a,b,c,d, x_weight,y_weight)` que calcula, en punto flotante, el mismo valor que se espera obtener del hardware. Esta función realiza:

1. Cálculo de los pesos de cada uno de los cuatro vecinos:

$$
\begin{aligned}
w_{00} &= (1 - x\_w) \cdot (1 - y\_w), \\
w_{10} &= x\_w \cdot (1 - y\_w), \\
w_{01} &= (1 - x\_w) \cdot y\_w, \\
w_{11} &= x\_w \cdot y\_w,
\end{aligned}
$$

donde x_w y y_w son las fracciones horizontales y verticales dentro de la celda.

2. Combinación lineal de los píxeles con sus pesos:

$$pix\_r = a \cdot w_{00} + b \cdot w_{10} + c \cdot w_{01} + d \cdot w_{11}.$$

3. Redondeo al entero más cercano con `$rtoi(pix_r + 0.5)` y saturación al rango [0, 255].

El valor entero que retorna esta función se utiliza como `expected` para comparar contra `pixel_out`.

2.4. Task `run_pixel` y cálculo de coordenadas

La lógica principal de prueba se encapsuló en una tarea automática `run_pixel(i_dst, j_dst, x_ratio, y_ratio)`. Esta tarea recibe las coordenadas del píxel destino y los *ratios* de escalamiento y realiza los siguientes pasos:

1. Calcula las coordenadas reales en la imagen fuente usando las mismas fórmulas del avance en Python:

$$x\_\text{src} = x\_ratio \cdot j\_\text{dst}, \qquad y\_\text{src} = y\_ratio \cdot i\_\text{dst}.$$

2. Obtiene las coordenadas enteras de los vecinos con `$floor` y `$ceil` para formar los índices `x_l`, `x_h`, `y_l` y `y_h`, y luego aplica un *clamp* para asegurar que se mantengan dentro de los límites de la imagen.

3. Calcula las fracciones:

$$x\_w = x\_\text{src} - x_l, \qquad y\_w = y\_\text{src} - y_l.$$

4. Lee los cuatro píxeles vecinos a, b, c y d del arreglo `image`.

5. Convierte las fracciones a formato Q0.8 generando `alpha_int` y `beta_int` mediante `x_w * 256.0 + 0.5` y `y_w * 256.0 + 0.5`, con redondeo y saturación al rango [0, 255]. Estos valores son exactamente los que se le entregan al DUT.

6. Llama a la función `bilinear_ref_pixel` para obtener el valor de referencia `expected`.

7. Muestra por consola toda la información útil del píxel destino (coordenadas, vecinos, pesos y valor esperado), lo que ayuda a depurar en caso de discrepancias.

2.5. Aplicación de estímulos al DUT y criterio de validación

Dentro de la misma tarea `run_pixel`, una vez preparados los datos, se realiza la secuencia de escritura hacia el DUT:

1. En un flanco de reloj se activa `valid_in` y se cargan los valores `I00`, `I10`, `I01`, `I11`, `alpha` y `beta` con los vecinos y pesos calculados.

2. En el siguiente flanco se desactiva `valid_in`.

3. El testbench espera a que `valid_out` sea 1 utilizando `wait (valid_out == 1'b1);`. En ese momento `pixel_out` ya contiene el resultado calculado por el hardware.

4. Se calcula la diferencia absoluta `diff` entre `pixel_out` y `expected`. Si `diff <= 1` se considera que el resultado está dentro de la tolerancia de ± 1 LSB y se incrementa `pass_count`. En caso contrario se incrementa `fail_count`.

Al final de la simulación se reporta un resumen de cuántos píxeles pasaron y cuántos fallaron. Si existe al menos un fallo, el testbench termina con `$fatal` para indicar que la prueba unitaria no fue satisfactoria. Además, se genera un archivo `.vcd` con las señales internas para inspeccionarlas en un visor de ondas.

3. Resultados de la verificación unitaria del modo secuencial

Para validar el funcionamiento del módulo `ModoSecuencial`, se ejecutó el testbench `tb_downscale_ModoSecuencial`, el cual compara cada píxel generado por el hardware contra un modelo de referencia en punto flotante implementado dentro del mismo testbench. En esta etapa se utilizó una imagen sintética de 4×4 píxeles, diseñada para facilitar la interpretación del proceso de interpolación. La imagen fuente utilizada fue:

10	30	50	70
90	110	130	150
170	190	210	230
240	245	250	255

El objetivo era generar una imagen destino de 3×3 píxeles, empleando interpolación bilineal con relaciones de escalamiento `x_ratio = 1.5` y `y_ratio = 1.5`, que son equivalentes a un factor de reducción de 2/3 tal como se usó en el modelo de Python del avance 1.

Para cada una de las nueve posiciones de la imagen destino, el testbench imprimió:

- Coordenadas reales x_{src} y y_{src} dentro de la imagen fuente,
- Píxeles vecinos (`a`, `b`, `c`, `d`),
- Pesos fraccionarios `x_w`, `y_w`,
- Valores en formato Q0.8 entregados al hardware (`alpha_int`, `beta_int`),
- Valor esperado según el modelo de referencia,
- Valor producido por el hardware,
- Diferencia absoluta entre ambos.

A continuación se muestra un resumen de los resultados obtenidos para cada píxel destino:

- Para todos los píxeles destino, la diferencia `diff` entre el resultado del hardware y el valor de referencia fue exactamente cero.
- No se presentaron casos de saturación incorrecta ni errores en el manejo del formato Q8.8.
- La generación de los pesos fraccionarios y su conversión a Q0.8 coincidieron con los valores esperados por el modelo.

Como ejemplo, el píxel destino $(1, 1)$ —que es el caso más representativo por tener fracciones horizontales y verticales de 0,5— obtuvo:

- Vecinos: $a = 110$, $b = 130$, $c = 190$, $d = 210$,
- Pesos: $x_w = 0{,}5$, $y_w = 0{,}5$,
- `alpha_int = 129`, `beta_int = 129`,
- Referencia: 160,
- Hardware: 160,
- Error: 0.

Finalmente, el testbench imprimió el siguiente resumen global:

```
=== FIN DOWNscale HW vs REF ===
Resumen: PASS=9, FAIL=0
TODOS los píxeles pasaron
```

Este resultado confirma que la implementación del modo secuencial es funcional, numéricamente correcta bajo el formato Q8.8, y consistente con el modelo de referencia en punto flotante dentro de una tolerancia de ± 1 LSB (la cual nunca fue alcanzada, ya que todos los casos coincidieron exactamente). Con esto se valida de manera completa el avance 3 del proyecto.

Listing 1: Resultados completos del testbench del modo secuencial

```
Imagen fuente 4x4:
  10 30 50 70
  90 110 130 150
  170 190 210 230
  240 245 250 255

Ratios: x_ratio=1.5000, y_ratio=1.5000

--- Pixel destino (0, 0) ---
  x_src=0.0000, y_src=0.0000
  x_l=0, x_h=0, y_l=0, y_h=0
  x_w=0.0000, y_w=0.0000
```

```
  a=10, b=10, c=10, d=10
  alpha_int=1 (0.0039), beta_int=1 (0.0039)
  Esperado (ref) = 10
  pixel_out = 10  (diff=0)
  PASS (dentro de +/- 1 LSB)

--- Pixel destino (0, 1) ---
  x_src=1.5000, y_src=0.0000
  x_l=1, x_h=2, y_l=0, y_h=0
  x_w=0.5000, y_w=0.0000
  a=30, b=50, c=30, d=50
  alpha_int=129 (0.5039), beta_int=1 (0.0039)
  Esperado (ref) = 40
  pixel_out = 40  (diff=0)
  PASS (dentro de +/- 1 LSB)

--- Pixel destino (0, 2) ---
  x_src=3.0000, y_src=0.0000
  x_l=3, x_h=3, y_l=0, y_h=0
  x_w=0.0000, y_w=0.0000
  a=70, b=70, c=70, d=70
  alpha_int=1 (0.0039), beta_int=1 (0.0039)
  Esperado (ref) = 70
  pixel_out = 70  (diff=0)
  PASS (dentro de +/- 1 LSB)

--- Pixel destino (1, 0) ---
  x_src=0.0000, y_src=1.5000
  x_l=0, x_h=0, y_l=1, y_h=2
  x_w=0.0000, y_w=0.5000
  a=90, b=90, c=170, d=170
  alpha_int=1 (0.0039), beta_int=129 (0.5039)
  Esperado (ref) = 130
  pixel_out = 130 (diff=0)
  PASS (dentro de +/- 1 LSB)

--- Pixel destino (1, 1) ---
  x_src=1.5000, y_src=1.5000
  x_l=1, x_h=2, y_l=1, y_h=2
  x_w=0.5000, y_w=0.5000
  a=110, b=130, c=190, d=210
  alpha_int=129 (0.5039), beta_int=129 (0.5039)
  Esperado (ref) = 160
  pixel_out = 160 (diff=0)
  PASS (dentro de +/- 1 LSB)

--- Pixel destino (1, 2) ---
```

```
  x_src=3.0000, y_src=1.5000
  x_l=3, x_h=3, y_l=1, y_h=2
  x_w=0.0000, y_w=0.5000
  a=150, b=150, c=230, d=230
  alpha_int=1 (0.0039), beta_int=129 (0.5039)
  Esperado (ref) = 190
  pixel_out = 190 (diff=0)
  PASS (dentro de +/- 1 LSB)

--- Pixel destino (2, 0) ---
  x_src=0.0000, y_src=3.0000
  x_l=0, x_h=0, y_l=3, y_h=3
  x_w=0.0000, y_w=0.0000
  a=240, b=240, c=240, d=240
  alpha_int=1 (0.0039), beta_int=1 (0.0039)
  Esperado (ref) = 240
  pixel_out = 240 (diff=0)
  PASS (dentro de +/- 1 LSB)

--- Pixel destino (2, 1) ---
  x_src=1.5000, y_src=3.0000
  x_l=1, x_h=2, y_l=3, y_h=3
  x_w=0.5000, y_w=0.0000
  a=245, b=250, c=245, d=250
  alpha_int=129 (0.5039), beta_int=1 (0.0039)
  Esperado (ref) = 248
  pixel_out = 248 (diff=0)
  PASS (dentro de +/- 1 LSB)

--- Pixel destino (2, 2) ---
  x_src=3.0000, y_src=3.0000
  x_l=3, x_h=3, y_l=3, y_h=3
  x_w=0.0000, y_w=0.0000
  a=255, b=255, c=255, d=255
  alpha_int=1 (0.0039), beta_int=1 (0.0039)
  Esperado (ref) = 255
  pixel_out = 255 (diff=0)
  PASS (dentro de +/- 1 LSB)

 === FIN DOWNscale HW vs REF ===
# Resumen: PASS=9, FAIL=0
# TODOS los pixeles pasaron
# ** Note: $finish
: C:/Users/sebas/OneDrive/Escritorio/Arqui2-
Proyecto/tb_downscale_ModoSecuencial.sv(239)
# Time: 245 ns  Iteration: 0
Instance: /tb_downscale_ModoSecuencial
```

4. Diseño teórico del manejo de memoria

En esta sección se describe el diseño teórico del subsistema de memoria del acelerador DSA para downscaling de imágenes. El objetivo principal es aprovechar únicamente los recursos internos de la FPGA (bloques M10K/BRAM), sin depender de una interfaz con SDRAM externa, y al mismo tiempo garantizar un flujo de datos eficiente para la interpolación bilineal.

4.1. Arquitectura general de memoria

El manejo de memoria se organiza alrededor de tres componentes principales:

- **Line buffers** o buffers de línea, que almacenan temporalmente las filas de la imagen fuente que se necesitan en cada instante.
- **Memory controller**, encargado de traducir coordenadas 2D de imagen a direcciones lineales y de orquestar las operaciones de lectura y escritura.
- **Output buffer**, que acumula los píxeles de la imagen reducida antes de escribirlos de vuelta en memoria.

El sistema está pensado para trabajar con imágenes en escala de grises de hasta 512×512 píxeles (8 bits por píxel). Esto implica:

- Imagen de entrada: $512 \times 512 = 262{,}144$ bytes (≈ 256 KB).
- Imagen de salida para factor 0.5: $256 \times 256 = 65{,}536$ bytes (64 KB).

La FPGA Cyclone V de la tarjeta DE1-SoC dispone de suficiente memoria interna para almacenar ambas imágenes y los buffers auxiliares asociados.

4.2. Line buffers y estrategia ping–pong

Concepto general

La interpolación bilineal requiere, para cada píxel de salida, el acceso a cuatro píxeles vecinos ubicados en dos filas consecutivas de la imagen fuente. En lugar de mantener la imagen completa en registros, se emplean *line buffers* que guardan únicamente las filas que están siendo utilizadas en el instante actual. Esto reduce el consumo de memoria rápida y permite un acceso más local a los datos.

Arquitectura de doble buffer

Se propone una estructura de doble buffer (ping–pong) compuesta por dos line buffers:

- Cada buffer almacena una fila completa de la imagen fuente: 512 píxeles $\times$ 8 bits $= 512$ bytes.

- La capacidad total de los line buffers es de 2×512 bytes $= 1024$ bytes (≈ 1 KB), fácilmente mapeable a un bloque M10K.
- Se utiliza memoria dual–port para permitir escritura de una fila mientras la otra se lee para el interpolador.

Operación ping–pong

La idea es ir reusando los mismos dos buffers a medida que el procesamiento avanza por la imagen:

1. **Primera iteración:** se cargan las filas 0 y 1 en Buffer 0 y Buffer 1, respectivamente, y se procesan los píxeles de salida que dependen de esas dos líneas.
2. **Segunda iteración:** la fila 2 se carga en Buffer 0 (sobrescribiendo la fila 0, que ya no se necesita) y ahora se procesan los píxeles utilizando las filas 1 y 2.
3. **Tercera iteración:** la fila 3 se carga en Buffer 1 (sobrescribiendo la fila 1), y así sucesivamente.

Con esta estrategia solo se requieren dos line buffers, independientemente del alto de la imagen fuente, lo que hace el diseño escalable y eficiente en términos de memoria.

Cada line buffer dispone de:

- Señales de escritura: `wr_en`, `wr_buf_sel`, `wr_addr`, `wr_data`.
- Señales de lectura: `rd_buf_sel`, `rd_addr` y salidas `rd_data0` y `rd_data1`, que permiten obtener simultáneamente los píxeles requeridos para la interpolación.

4.3. Controlador de memoria

Función principal

El *memory controller* actúa como interfaz entre la FSM de control y los recursos de memoria. Su tarea es convertir operaciones de alto nivel del tipo "leer fila N" o "escribir línea de salida M" en direcciones y accesos secuenciales sobre la memoria interna.

Direccionamiento 2D

Para cada píxel se parte de una coordenada $(\text{fila}, \text{columna})$ y se calcula una dirección lineal mediante:

$$\text{addr} = \text{base} + (\text{fila} \times \text{ancho} + \text{columna}) \times \text{bytes\_por\_píxel}.$$

Por ejemplo, para una imagen de 512×512, el píxel en $(100, 50)$ se encuentra en:

$$\text{addr} = \text{base} + (100 \times 512 + 50) \times 1 = \text{base} + 51{,}250.$$

Lectura de líneas

La lectura de una fila completa se realiza como un *burst* de 512 ciclos (un píxel por ciclo). La secuencia típica es:

1. La FSM indica que se debe leer la fila N.
2. El controlador calcula la dirección inicial correspondiente.
3. Se generan 512 lecturas consecutivas y cada byte se escribe en el line buffer seleccionado.
4. Al terminar, se envía una señal de "línea lista" a la FSM.

Escritura de la imagen de salida

De forma análoga, cuando el *output buffer* contiene una línea completa de la imagen reducida (por ejemplo, 256 píxeles para un factor de escala de 0.5), el controlador realiza un *burst write* para almacenar esos datos en la región de memoria reservada para la imagen de salida.

4.4. Output buffer

El *output buffer* es un arreglo lineal que acumula los píxeles generados por el interpolador antes de escribirlos en memoria. Para el caso de downscaling a la mitad, una línea de salida tiene 256 píxeles, por lo que el buffer requiere 256 bytes.

El flujo de datos es el siguiente:

1. El interpolador bilineal produce un píxel de salida por ciclo en modo secuencial.
2. Cada valor se almacena en la posición correspondiente del *output buffer*.
3. Cuando la línea está completa, la FSM activa la operación de escritura y el *memory controller* vuelca el contenido del buffer a memoria mediante un burst.
4. Una vez escrita la línea, el buffer se reutiliza para la siguiente.

Este esquema permite que las escrituras a memoria se hagan de forma agrupada, lo cual es más eficiente que escribir píxel por píxel.

4.5. Coordinación mediante FSM

Todo el flujo de lectura, procesamiento y escritura es coordinado por una máquina de estados finita (FSM) que recorre, de forma simplificada, los siguientes estados:

- **IDLE**: espera de configuración inicial y señal de inicio.
- **SETUP**: inicializa direcciones base, contadores y parámetros de imagen.
- **READ_LINE0** y **READ_LINE1**: cargan el par de filas necesario en los line buffers.

- **PROCESS**: habilita el interpolador para generar los píxeles de salida utilizando los datos de los line buffers.
- **WRITE_OUT**: escribe la línea procesada desde el *output buffer* a memoria.
- **NEXT_LINE**: actualiza índices de fila y realiza el intercambio ping–pong de los buffers.
- **DONE**: indica que toda la imagen ha sido procesada.

Durante estos estados, la FSM genera señales de control como `mem_read_en`, `mem_write_en`, `line_buf_wr_en`, `line_buf_swap`, `interp_enable` y `output_valid`, que sincronizan el comportamiento de todos los bloques.

4.6. Análisis de recursos y rendimiento

En términos de memoria interna, el diseño requiere:

- **Line buffers**: 2×512 bytes ≈ 1 KB (pueden mapearse a un bloque M10K).
- **Output buffer**: 256 bytes adicionales (otro bloque M10K).
- **Imágenes**: 256 KB para la imagen de entrada y 64 KB para la imagen de salida.

La DE1-SoC cuenta con suficientes bloques M10K/BRAM para este escenario. Para imágenes mayores a 512×512 sí sería necesario considerar una interfaz con SDRAM externa, pero esa opción queda como extensión opcional.

En cuanto al número de ciclos, para una transformación de 512×512 a 256×256 en modo secuencial se obtiene, de forma aproximada:

- Lectura de líneas: 512 ciclos $\times$256 líneas $\approx$ 131,072 ciclos.
- Procesamiento: 256 píxeles $\times$256 líneas $\times$ latencia efectiva del interpolador (del orden de pocos ciclos), del orden de cientos de miles de ciclos.
- Escritura de salida: 256 ciclos $\times$256 líneas = 65,536 ciclos.

A una frecuencia de 50 MHz, el tiempo de procesamiento total se mantiene en el orden de decenas de milisegundos, lo cual es razonable para un acelerador dedicado de propósito específico.

4.7. Consideraciones para el modo SIMD

Cuando se extienda la arquitectura al modo SIMD (por ejemplo, procesando 4 píxeles por ciclo), el diseño de memoria se debe adaptar, pero sin cambiar la idea base:

- Los line buffers necesitarán ofrecer más anchos de lectura para poder entregar simultáneamente los 4×4 píxeles vecinos requeridos por el vector SIMD.

- El *memory controller* podrá agrupar aún más los accesos en bursts, sacando mejor provecho del ancho del bus interno.
- El *output buffer* deberá ser capaz de recibir varios píxeles por ciclo desde la unidad SIMD.

En términos de rendimiento, al cuadruplicar el número de píxeles procesados por ciclo se espera una reducción cercana al 75 % en el tiempo dedicado exclusivamente a la etapa de interpolación, manteniendo la misma estructura global de memoria.

4.8. Conclusiones

El diseño de memoria propuesto se basa en una arquitectura de line buffers con estrategia ping–pong, un *memory controller* que abstrae el direccionamiento 2D y un *output buffer* que permite escrituras agrupadas. Esta combinación aprovecha de forma eficiente los bloques M10K/BRAM de la FPGA y limita el número de accesos necesarios para la interpolación bilineal.

Además, la arquitectura es escalable: con pequeños ajustes en los anchos de lectura y en el *datapath* es posible migrar del modo secuencial al modo SIMD manteniendo la misma organización básica de memoria. Para resoluciones mayores a 512×512 se podría incorporar SDRAM externa siguiendo la misma lógica de line buffers, como una extensión futura del sistema.

5. Resumen del avance

En este tercer avance se desarrolló e implementó el modo secuencial del acelerador DSA para downscaling mediante interpolación bilineal, y además se planteó el diseño teórico del manejo de memoria interna que utilizará el sistema en etapas posteriores. El objetivo principal de esta etapa era validar el correcto funcionamiento del *datapath* aritmético en punto fijo y, al mismo tiempo, definir una organización de memoria compatible con la arquitectura propuesta y con las restricciones de la FPGA.

Desde el punto de vista de cómputo, se diseñó el módulo `ModoSecuencial`, el cual recibe los cuatro píxeles vecinos (`I00`, `I10`, `I01`, `I11`) junto con los pesos fraccionarios `alpha` y `beta` en formato Q0.8, y calcula un píxel interpolado utilizando la fórmula bilineal dividida en una interpolación horizontal seguida de otra vertical. El módulo implementa todas las operaciones internas en Q8.8, incluyendo diferencias con signo, productos en Q16.16, recorte de bits, redondeo y saturación a 8 bits para generar la salida final. La interfaz incorpora también señales de control (`clk`, `rst`, `valid_in`, `valid_out`) para garantizar que se produzca un píxel por ciclo en modo secuencial.

Para validar este módulo se desarrolló un testbench unitario, que reproduce el algoritmo de interpolación bilineal en punto flotante y compara cada resultado del hardware contra una versión de referencia. El testbench utiliza una imagen sintética de 4×4 píxeles y genera una imagen destino de 3×3, calculando coordenadas reales, vecinos, pesos, valores Q0.8 y el resultado esperado para cada píxel. Para cada caso se evalúa la diferencia entre el valor del hardware y el modelo de referencia, aceptando una tolerancia máxima de ± 1 LSB.

Los resultados mostraron que los nueve píxeles de salida coincidieron exactamente con el modelo, con `PASS=9` y `FAIL=0`, lo que confirma que el modo secuencial está correctamente implementado. El log completo de la simulación se incluye como *listing* en la sección de resultados.

Además del *datapath* secuencial, en este avance se planteó el diseño teórico del manejo de memoria interna. Se definió una arquitectura basada en:

- **Line buffers** con estrategia ping–pong para almacenar únicamente las dos filas de la imagen fuente necesarias en cada instante, reduciendo el uso de BRAM.
- Un **memory controller** encargado de mapear coordenadas 2D a direcciones lineales y de realizar lecturas y escrituras en ráfagas (*bursts*) sobre la memoria interna de la FPGA.
- Un **output buffer** que acumula una línea completa de la imagen reducida antes de escribirla en memoria, mejorando la eficiencia de las escrituras.

Este planteamiento considera explícitamente las dimensiones máximas de la imagen (512×512), el uso exclusivo de bloques M10K/BRAM para almacenar tanto la imagen de entrada como la de salida, y la futura extensión al modo SIMD (donde se procesarán varios píxeles por ciclo). Aunque en este avance la lógica de memoria todavía no se implementa en hardware, la definición de la arquitectura de line buffers, controlador de memoria y *output buffer* deja una base clara para integrar el *datapath* secuencial dentro de un flujo completo de lectura–procesamiento–escritura en los siguientes avances.

Referencias

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Elsevier, 2017.

[2] Project F, "Fixed Point Numbers in Verilog," *Project F Blog*, 2025. [Online]. Available: https://projectf.io/posts/fixed-point-numbers-in-verilog/

[3] D. W. Gisselquist, "Strategies for Pipelining," *ZipCPU Blog*, 2017. [Online]. Available: https://zipcpu.com/blog/2017/08/14/strategies-for-pipelining.html

[4] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/optimization-manual.html

[5] A. Benavides, "Guía JTAG para Comunicación FPGA-PC," *GitHub Repository*, 2025. [Online]. Available: https://github.com/Abner2111/GuiaJtag